An Exercise in Transforming Wijngaarden Grammars into Knuthian Grammars (Mathematical Methods in Software Science and Engineering : Second Conference)

AUTHOR(S):

TOKUDA, TAKEHIRO

CITATION:

TOKUDA, TAKEHIRO. An Exercise in Transforming Wijngaarden Grammars into Knuthian Grammars (Mathematical Methods in Software Science and Engineering : Second Conference). 数理解析研究所講究録 1980, 396: 289-309

ISSUE DATE:

1980-09

URL:

http://hdl.handle.net/2433/105018

数理解析研究所講究録
396 巻 1980 年 289-309

An Exercise in Transforming Wijngaarden Grammars into Knuthian Grammars

Takehiro Tokuda
Department of Information Science
Tokyo Institute of Technology
Ookayama, Meguro, Tokyo 152, Japan

1. Introduction

The purpose of this paper is to show that a widespread negative view [1, 6, 7, 8, 16, 21] of Wijngaarden grammars [20] (also known as two-level grammars; henceforth referred to as W-grammars) among computer scientists is not necessarily appropriate for actual programming languages. More precisely, the prevailing view may be stated as follows:

"W-grammars are essentially context-free grammars (CFG) being attached to attribute equations. Even in a toy W-grammar, present techniques for solving such equations are not powerful enough to treat them mechanically. Hence, writing a specification in a W-grammar is a waste of time."

Our objection to the above view is as follows:

(1) Although the mechanical translatability of W-grammars into procedural languages is negative in principle, it is also true that we can usually convert the definition of programming languages given in W-grammars into that of attribute grammars [13] (also known as Knuthian grammars and K-grammars) in a semi-automatic manner;

(2) Features of W-grammars viewing as a specification language are less appreciated among computer scientists. We believe we should encourage the use of W-grammars as an explicit specification language in various applications.

We shall elaborate these points later by illustrating many small but essential examples.

The organization of this paper is as follows. In section 2, we explain several features of W-grammars as a specification language. In section 3, we briefly compare features of attribute

grammars with that of W-grammars. In section 4, we give a general strategy to transform a specification given in W-grammars into that of attribute grammars, then we give two elementary transformation examples. In section 5, we consider an advanced problem which seems difficult to transform by the strategy given in section 4. In section 6, we give conclusions.

In what follows, we assume that the reader is familiar with basic terminology of W-grammars and K-grammars (see [4, 15], for example).

2. W-grammars as a specification language

This section presents an illustration of features of W-grammars when we regard W-grammars as a specification language.

Problem A. (Negation is not easy)

A-1 Construct a predicate EQUAL such that EQUAL is true, if two given characters are equal; EQUAL is undefined, otherwise.

A-2 Construct a predicate NOT_EQUAL such that NOT_EQUAL is true, if two given characters are different; NOT_EQUAL is undefined, otherwise.

Solution.

A-1 ALPHA :: a; b; c; d; e; f; g; h; i; j; k; l; m; n; o; p; q;
r; s; t; u; v; w; x; y; z.

EMPTY :: .

ALPHA is equal to ALPHA : EMPTY.

A-2 STRING :: ALPHA STRING; EMPTY.

ALPHA1 is not equal to ALPHA2 :

(STRING1 ALPHA1 STRING2 ALPHA2 STRING3)

is equal to (abcdefghijklmnopqrstuvwxyz).

(STRING) is equal to (STRING) : EMPTY.

Problem B. (Disjunction and Conjunction are easy)

B-1 Construct a predicate EQUAL_AND_NOT_EQUAL equivalent to EQUAL and NOT_EQUAL.

B-2 Construct a predicate EQUAL_OR_NOT_EQUAL equivalent to EQUAL or NOT_EQUAL.

Solution.

B-1 ALPHA1 is equal to and not equal to ALPHA2 :

ALPHA1 is equal to ALPHA2,

ALPHA1 is not equal to ALPHA2.

B-2 ALPHA1 is equal to or not equal to ALPHA2 :

ALPHA1 is equal to ALPHA2;

ALPHA1 is not equal to ALPHA2.

Problem C. (Existential quantifier and Recursion are easy)

C-1 Construct a predicate AT_LEAST_ONE such that AT_LEAST_ONE is true, if the following equation

$$X^3 + 11X = 6X^2 + 6$$

has at least one positive integer solution; AT_LEAST_ONE is undefined, otherwise.

Solution.

C-1 VAL :: one VAL; one.

the equation has at least one positive integer solution :

VAL1 VAL2 is equal to VAL3 one one one one one one,

```
        VAL1 is cubic of VAL,
        VAL2 is equal to
             VAL VAL VAL VAL VAL VAL VAL VAL VAL VAL VAL,
        VAL3 is equal to VAL4 VAL4 VAL4 VAL4 VAL4 VAL4,
        VAL4 is square of VAL.
    VAL is equal to VAL : EMPTY.
    VAL1 VAL VAL one is square of VAL one :
        VAL1 is square of VAL.
    one is square of one : EMPTY.
    VAL1 VAL2 VAL2 VAL2 VAL VAL VAL one is cubic of VAL one :
        VAL1 is cubic of VAL,
        VAL2 is square of VAL.
    one is cubic of one : EMPTY.
```

Problem D. (Universal quantifier over finite domain is not difficult)

D-1 Construct a predicate PRIME such that PRIME is true, if a given positive integer is a prime number; PRIME is undefined, otherwise.

Solution.

```
D-1 VAL one is prime :
        VAL one is equal to one one;
        VAL one is equal to one one VAL1,
        VAL one is checked by one one to VAL.
    VAL one is checked by VAL1 to VAL :
        VAL1 VAL2 is equal to VAL,
        VAL one is not divisible by VAL1,
        VAL one is checked by VAL1 one to VAL.
```

VAL one is checked by VAL to VAL :

VAL one is not divisible by VAL.

VAL1 VAL is not divisible by VAL :

VAL1 is not divisible by VAL.

VAL1 is not divisible by VAL :

VAL1 VAL2 is equal to VAL.

VAL is equal to VAL : EMPTY.

Problem E. (The use of global context is easy; moreover, reverse recursive thinking is possible)

Construct a symbol table of a simple block structured programming language.

Solution.

E-1 DEC :: identifier ALPHA has INTBOOL type.

DECS :: DEC DECS; DEC.

NEST :: new EMPTY; NEST new DECS.

intbool :: integer; bool.

ALPHA :: a; b; c; d; e; f; g; h; i; j; k; l; m; n; o; p; q; r; s; t; u; v; w; x; y; z.

new EMPTY program :

begin symbol,

declaration train with DECS,

go on symbol,

new EMPTY new DECS statement train,

end symbol.

declaration train with DEC DECS :

declaration with DEC,

go on symbol,

declaration train with DECS.

declaration train with DEC :

declaration with DEC.

declaration with identifier ALPHA has INTBOOL type :

INTBOOL symbol,

letter ALPHA symbol.

NEST statement train :

NEST statement,

go on symbol,

NEST statement train.

NEST statement train :

NEST statement.

NEST statement :

NEST non-block statement.

NEST statement :

begin symbol,

declaration train with DECS,

go on symbol,

NEST new DECS statement train,

end symbol.

From the above examples, we may safely say that the simple mechanism of W-grammars is quite powerful to most neatly describe recursively enumerable sets.

3. Attribute grammars as a procedural language

In this section we briefly review features of K-grammars [2, 3, 11, 12, 13] taking account of features of W-grammars. In order to introduce our notation of K-grammars in this paper, we first give a classical example of K-grammars.

Example. (The value of a binary integer)

Scale : Inherited, Value : Synthesized.

1) number : digit series.
   Scale(digit series):= 0,
   Value(number):= Value(digit series).
2) digit series1 : digit series2, digit.
   Scale(digit):= Scale(digit series1),
   Scale(digit series2):= Scale(digit series1) + 1,
   Value(digit series1):= Value(digit series2) + Value(digit).
3) digit series : digit.
   Scale(digit):= Scale(digit series),
   Value(digit series):= Value(digit).
4) digit : one symbol.
   Value(digit):= 2 ** Scale(digit).
5) digit : zero symbol.
   Value(digit):= 0.

As you can easily notice from the above example, attribute grammars may be considered as a restricted subclass of W-grammars in the following sense.

(1) Attributes are classified into inherited attributes and synthesized attributes disjointedly.

(2) The order of evaluation of attributes is determined locally to each production. Hence, if the attribute grammar has no circular definition, the topological sorting of a dependency ordering is enough to determine every value of attributes.

(3) Once the value has been defined, there will be no change of value. Moreover, by any evaluation order, the ultimately defined value is unique.

Therefore it is important to give a general practical method to transform W-grammars into K-grammars. We will describe such a method in Section 4.

4. W-grammars as almost K-grammars

We first describe the outline of general strategy to transform a naturally written Wijngaarden specification to that of K-grammars. Then we give two elementary transformation examples.

[General Strategy]

(1) [Extract CFG portions from given Hyperrules] Try to separate METANOTIONS from Hyperrules, and regard the remaining portions as extracted CFG. Regard these METANOTIONS as attributes associated with those CFG. In the case that a METANOTION has a finite domain, we must choose to either merge it in CFG or separate it as an attribute.

(2) [One pass generation of protonotions] Using oracle generation of protonotions and check of relations among these protonotions, establish semantic functions of attributes

where every value of attributes (protonotions) is determined in a top-down way.

(3) [Introduction of synthesized attributes and additional inherited attributes] If there is an attribute whose value is determined at some level of CFG production, then try to introduce a synthesized attribute whose value will be carried up successively until the most recent point of oracle generation. If necessary, introduce additional inherited attributes at the point of oracle generation. Repeat the process until there are no oracle generation.

Problem F.

Generate the following language

$$\{a^n b^n c^n \mid n \geq 1\}.$$

Solution.

F-1 PROD ::

large s becomes small a large s large b large c end;
large s becomes small a small b large c end;
large c large b becomes large b large c end;
small b large b becomes small b small b end;
small b large c becomes small b small c end;
small c large c becomes small c small c end.

PRODS :: PROD PRODS; PROD.

STRING :: large ALPHA STRING; small ALPHA STRING; EMPTY.

SMALLSTRING :: small ALPHA SMALLSTRING; EMPTY.

THING :: ALPHA THING; EMPTY.

ALPHA :: a; b; c; d; e; f; g; h; i; j; k; l; m; n; o; p; q;
r; s; t; u; v; w; x; y; z.

```
EMPTY :: .
start : SMALLSTRING,
     where SMALLSTRING is from large s via PRODS series.
where STRING3 is from STRING1 via PROD PRODS series :
     where STRING2 is from STRING1 via PROD series,
     where STRING3 is from STRING2 via PRODS series.
where STRING2 is from STRING1 via PROD series :
     where (STRING1) is (STRING5 STRING3 STRING6),
     where (STRING2) is (STRING5 STRING4 STRING6),
     where (PROD) is (STRING3 becomes STRING4 end).
where (THING) is (THING) : EMPTY.
small ALPHA SMALLSTRING :
     letter ALPHA symbol, SMALLSTRING.
F-2 PRODUCTION ::
     large s becomes small a large a small b large b small
     c large c end;
     large a becomes small a large a end;
     large b becomes small b large b end;
     large c becomes small c large c end;
     large a becomes end;
     large b becomes end;
     large c becomes end.
PRODUCTIONS :: PRODUCTION PRODUCTIONS; PRODUCTION.
LOOP :: large a becomes small a large a end large b
     becomes small b large b end large c becomes small c
     large c end LOOP; EMPTY.
CONTROLLED :: large s becomes small a large a small b
     large b small c large c end LOOP large a becomes end
```

large b becomes end large c becomes end.

start : SMALLSTRING,

where SMALLSTRING is from large s

via PRODUCTIONS series,

where (PRODUCTIONS) is (CONTROLLED).

F-3 VALUE :: one VALUE; one.

EACH :: a; b; c.

start : VALUE a series, VALUE b series, VALUE c series.

one VALUE EACH series :

letter EACH symbol, VALUE EACH series.

one EACH series : letter EACH symbol.

Problem F'.

Transform the above W-grammar F-3 into a K-grammar.

Solution.

[1] 1) start : a series, b series, c series.

2) a series1 : letter a symbol, a series2

3) a series : letter a symbol.

4) b series1 : letter b symbol, b series2.

5) b series : letter b symbol.

6) c series1 : letter c symbol, c series2.

7) c series : letter c symbol.

[2] Value : Inherited, Bool : Synthesized.

1) Value(a series):= m1, Value(b series):= m2,

Value(c series):= m3,

Bool(start):= true, if m1 = m2 = m3,

Bool(start):= false, otherwise.

2) Value(a series2):= Value(a series1) - 1.

3) (Value(a series) = 1.)

4) Value(b series2):= Value(b series1) - 1.

5) (Value(b series) = 1.)

6) Value(c series2):= Value(c series1) - 1.

7) (Value(c series) = 1.)

[3] Value, Bool : Synthesized.

1) Bool(start):= true, if Value(a series) =
Value(b series) = Value(c series),
Bool(start):= false, otherwise.

2) Value(a series1):= Value(a series2) + 1.

3) Value(a series):= 1.

4) Value(b series1):= Value(b series2) + 1.

5) Value(b series):= 1.

6) Value(c series1):= Value(c series2) + 1.

7) Value(c series):= 1.

Problem E'.

Transform the W-grammar E-1 into a K-grammar.

Solution.

[1] 1) program : begin symbol, declaration train,
go on symbol, statement train, end symbol.

2) declaration train1 : declaration, go on symbol,
declaration train2.

3) declaration train : declaration.

4) declaration : integer symbol, letter a symbol.
declaration : integer symbol, letter b symbol.
...

5) statement train1 : statement, go on symbol,

statement train2.

6) statement train : statement.

7) statement : non-block statement.

8) statement : begin symbol, declaration train,
go on symbol, statement train, end symbol.

[2] Dec, Nest : Inherited.

1) Dec(declaration train):= d,
Nest(statement train):= new new d.

2) Dec(declaration):= First(Dec(declaration train1)),
Dec(declaration train2):=
Rest(Dec(declaration train1)).

3) Dec(declaration):= Dec(declaration train).

4) (Dec(declaration) = identifier a has integer type,
...
)

5) Nest(statement):= Nest(statement train1),
Nest(statement train2):= Nest(statement train1).

6) Nest(statement):= Nest(statement train).

7) Nest(non-block statement):= Nest(statement).

8) Dec(declaration train):= d,
Nest(statement train):= Nest(statement) new d.

[3] Dec : Synthesized, Nest : Inherited.

1) Nest(statement train):= new new Dec(declaration train).

2) Dec(declaration train1):=
Dec(declaration) Dec(declaration train2)

3) Dec(declaration train):= Dec(declaration).

4) Dec(declaration):= identifier a has integer type,
... .

5) Nest(statement):= Nest(statement train1),

Nest(statement train2):= Nest(statement train1).

6) Nest(statement):= Nest(statement train).

7) Nest(non-block statement):= Nest(statement).

8) Nest(statement train):=

Nest(statement) new Dec(declaration train).

As you see from these examples, the transformation strategy we illustrated here is heavily dependent on the possibility of changing the reverse recursive description into a normal bottom-up description.

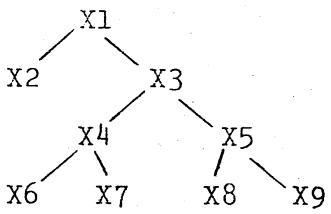
5. An Advanced Problem

In this section we consider an advanced problem naturally arisen in the semantic definition of Balancing in Algol 68 [20] or Overloading [5, 9] in Ada [10, 19]. It is not necessarily easy to describe neat semantic definition of these features in terms of K-grammars, since the nature of the features seems combinatorial.

For the sake of simple presentation, we consider the problem in the following setting.

Problem G.

We can assign either 1 or 2 or 3 to each node of the following labeled directed tree T



according the following permissible assignment relation R

(X1; X2; X3) = (2; 1; 1) or (1; 3; 2),

(X3; X4; X5) = (1; 2; 1) or (1; 1; 2) or (2; 1; 3),

(X4; X6; X7) = (1; 1; 2) or (2; 1; 2) or (3; 1; 2),

(X5; X8; X9) = (1; 1; 2) or (2; 1; 2).

G-1 Determine whether or not the tree T has at least one assignment to all the nodes under the restriction R.

G-2 Determine whether or not the tree T has exactly one assignment to all the nodes under the restriction R.

We summarize our observations to this type of problem.

(1) It is straightforward to write a specification of the problem G-1 in a W-grammar. (However, it is very tedious to write G-2 in terms of a W-grammar.)

Solution.

G-1 VAL :: VAL one; one.

start : VAL x1

one one x1 : one x2 symbol, one x3.

one x1 : one one one x2 symbol, one one x3.

one x3 : one one x4, one x5.

one x3 : one x4, one one x5.

one one x3 : one x4, one one one x5.

one x4 : one x6 symbol, one one x7 symbol.

one one x4 : one x6 symbol, one one x7 symbol.

one one one x4 : one x6 symbol, one one x7 symbol.

one x5 : one x8 symbol, one one x9 symbol.

one one x5 : one x8 symbol, one one x9 symbol.

(2) Three pass evaluation strategy is sufficient for the problem G-2 in the following sense: We first topologically sort the permissible assignment relations. Then we evaluate from bottom to top according to the topological ordering. Then we evaluate again from top to bottom according to the reverse topological ordering. Thus we obtain the set of all solutions at each node. (We can prove the validity of this algorithm by the induction on the size of a problem, i.e. the height of the tree.)

Solution.

G-2 Bool, S : Synthesized, i : Inherited.

Root(i,j,k) : Function

Root is initially empty;

for each relation (ti; tj; tk) at Xi, Xj and Xk

do if (Xj is a leaf or tj is in S(Xj)) and

(Xk is a leaf or tk is in S(Xk))

then add ti to Root fi

od.

Leaf(i,j,k,l) : Function

Leaf is initially empty;

for each relation (ti; tj; tk) at Xi, Xj and Xk

do if ((Xi is the root and ti is in S(Xi)) or

(Xi is not the root and ti is in I(Xi)))
and (Xj is a leaf or tj is in S(Xj)) and
(Xk is a leaf or tk is in S(Xk))
then add t1 to Leaf fi

od.

1) x1 : x2 symbol, x3.
S(x1):= Root(1,2,3),
I(x2):= Leaf(1,2,3,2),
I(x3):= Leaf(1,2,3,3),
Bool(x1):= (S(x1) and I(x2) are single sets) and Bool(x3).

2) x3 : x4, x5.
S(x3):= Root(3,4,5),
I(x4):= Leaf(3,4,5,4),
I(x5):= Leaf(3,4,5,5),
Bool(x3):= (I(x3) is a single set) and
Bool(x4) and Bool(x5).

3) x4 : x6 symbol, x7 symbol.
S(x4):= Root(4,6,7),
I(x6):= Leaf(4,6,7,6),
I(x7):= Leaf(4,6,7,7),
Bool(x4):= (I(x4), I(x6) and I(x7) are single sets).

4) x5 : x8 symbol, x9 symbol.
S(x5):= Root(5,8,9),
I(x8):= Leaf(5,8,9,8),
I(x9):= Leaf(5,8,9,9),
Bool(x5):= (I(x5), I(x8) and I(x9) are single sets).

(3) If we consider the problem G-1 on an undirected graph T in general, then this problem becomes NP-complete when we

consider the size of the problem as the number of nodes of T. The k-colorability problem of T can be coded as follows. If nodes X1, ..., Xn are neighbors of node X0, then add permissible assignment relation:

(X0; X1; ...; Xn) = (i0; i1; ...; in) for integers i0, i1, ..., in from 1 to k such that i1, ..., in are not equal to i0.

This completes our observations of this type of problem.

6. Conclusions

We have shown that W-grammars have nice features when we explicitly use W-grammars as a specification language. Then we have shown that a naturally written W-grammar specification can be transformed into a K-grammar definition in a semi-automatic way. Finally we mentioned a type of problem whose specification in a W-grammar cannot be easily transformed into a K-grammar according to the transformation strategy we presented.

References

[1] Baker, J.L.: Grammars with structured vocabulary: a model for the Algol 68 Definition, Information and Control 20:4, 351-395 (1972)

[2] Bochmann, G.V.: Semantic evaluation from left to right,

Comm. ACM 19:2, 55-62 (1976)

[3] Chirica, L.M., Martin, D.F.: An order-algebraic definition of Knuthian semantics, Mathematical System Theory 13, 1-27 (1979)

[4] Cleaveland, J., Uzgalis, R.: Grammars for programming languages, American Elsevier, N.Y., 1976

[5] Dausmann, M., et al.: Overloading in ADA, TR No.23/79, University of Karlsruhe, 1979

[6] Dembinski, P., Maluszynski, J.: Attribute grammars and two-level grammars: a unifying approach, Lecture Notes in Computer Science 64, 143-154 (1978)

[7] Deussen, P.: A decidability criterion for Van Wijngaarden grammars, Acta Informatica 5, 353-375 (1975)

[8] Deussen, P., Mehlhorn, K.: Van Wijngaarden grammars and space complexity class EXSPACE, Acta Informatica 8, 193-199 (1977)

[9] Ganzinger, H., Ripken, K.: Operator identification in Ada: formal specification, complexity, and concrete implementation, SIGPLAN Notices 15:2, 30-42 (1980)

[10] Ichbiah, J.D., et al.: Rationale for the design of the ADA programming language, SIGPLAN Notices 14:6 (1979)

[11] Jazayeri, M., Ogden, W.F., Rounds, W.C.: The intrinsically exponential complexity of the circularity problem for attribute grammars, Comm. ACM 18:12, 697-706 (1975)

[12] Kennedy, K., Warren, S.K.: Automatic generation of efficient evaluators for attribute grammars, Conf, Rec. 3rd ACM Symp. POPL, 32-49 (1976)

[13] Knuth, D.E.: Semantics of context-free languages,

Mathematical System Theory 2, 127-145 (1968)

[14] Koster, C.H.A.: Affix grammars, Algol 68 Implementation, North-Holland, Amsterdam, 1971

[15] Marcotty, M., Ledgard, H.F., Bochmann, G.V.: A sampler of formal definitions, Computing Surveys 8:2 (1976)

[16] Sintzoff, M.: Existence of a Van Wijngaarden syntax for every recursively enumerable set, Annale de la Societe Scientifigue de Bruxelles 81, 115-118 (1967)

[17] Tokuda, T., Tokuda, J., Sassa, M., Inoue, K.: Metanotion chart for revised Algol 68, SIGPLAN Notices 12:[illegible], 11-14 (1977)

[18] Tokuda, T.: Wijngaarden grammars as Knuthian grammars, Proc. 20th Ann. Conf. IPSJ, 207-208 (1979)

[19] United States Department of Defense: Preliminary ADA reference manual, SIGPLAN Notices 14:6 (1979)

[20] Van Wijngaarden, A., et al.: Revised report on the algorithmic language Algol 68, Springer-Verlag, Berlin, 1976

[21] Wegner, L.: Bracketed two-level grammars - a decidable and practical approach to language definitions, Lecture Notes in Computer Science 71, 668-682 (1979)